Every day we use a subway or a bus to go to work, university, or school. At lunchtime, we visit a cafe or a restaurant. Very rarely do we have problems with these and other services that help organize our daytime. Such existing agreements are helping people to facilitate their basic needs and make life predictable in a good way. Let's see how we can create similar agreements between programming objects.

**Basic definitions**

Skills define what work you can do. If you are qualified for some job, you can sign a contract and take it! We can say that a contract is a legal agreement that people have. The name for such legal agreements in programming is interfaces.

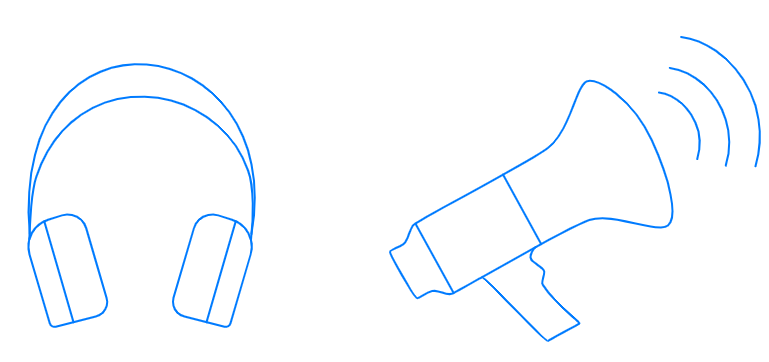
An**interface** is a collection of methods that describes the behavior of an object. To implement the interface, an object should implement all the methods from it. The behavior of an object is stiff, and it cannot change during the execution of a program. That's the difference between people and objects.

With the interface implementation, you can be sure that an object can do the job it accepts. You can also choose which object will do the work because they can do it a bit differently.

**One-method interface**

When you're at home, you may prefer to listen to music with loudspeakers, but in public, you hopefully use headphones. We can say that loudspeakers and headphones implement the interface with a method MAKE\_SOUND.

It's sensible to choose suitable names for our interfaces. In our case, SOUNDMAKER works just fine. The naming practices will vary from language to language, but commonly, we use verbs for method names and nouns or adjectives for interface names.

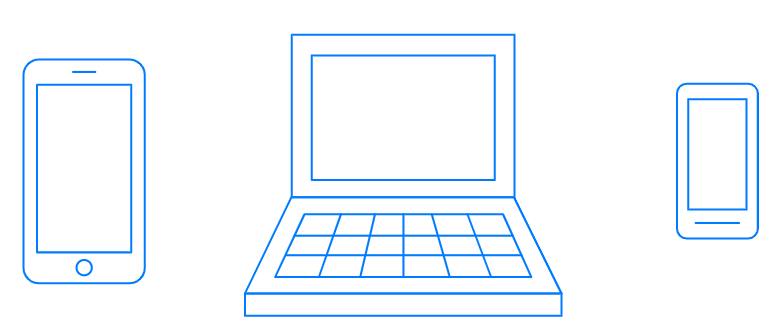


If you want to implement an interface with a method READ, you can call it READER or READABLE. From the verb *transform,* you can create names TRANSFORMER and TRANSFORMABLE. This convenient way doesn't always work, so sometimes you'd have to come up with a different name that makes sense.

One method stands for only one skill of an object, but in many situations, this is enough to make an interface effective.

**Complex interface**

To play music, you also need a device to store and manage audio files, because speakers and headphones only output the sound. You can use a laptop, a smartphone, or a player: all of them implement the interface AUDIOPLAYER.



What methods do you think this interface should have? We can choose the minimum amount of methods a device would need to play music:



Notice that our methods resemble control elements of an actual player. This makes our code clear because by looking at this interface, we understand what the object that implements the interface is doing.

There is no preferential difference between one-method and complex interfaces, but if you go with complex, the question is: how big can they get?

**Responsibility of an interface**

To answer the size question, it's not about the number of methods but the responsibility an object takes. When you use a one-method interface, there's a big chance that there are plenty of objects in your code that can do the work. As your interface grows, your objects should obtain more and more skills. Can our objects cope with so many responsibilities?

For a second, imagine a world with only one type of device that can play music, one way to play the sound, and where you can listen to music only in special places because someone creates interfaces with a lot of responsibilities for them. Do we want to build a world like this in our code?

We want our objects to communicate through simple interfaces even when they have a lot of methods. All of these methods should be *necessary* to do the job right. If there is no need for some methods, we can move them to a new interface, or even remove them completely. For example, we can add to our AUDIOPLAYER a method REWIND, but there is no need to have a method RECORD there because it is not related to playing music.

If you don't break up interfaces with many responsibilities into smaller ones, the code will become tangled, and neither you, nor other developers will understand it.

**Conclusion**

In this topic, we've covered interfaces. We can define one or more methods for an interface, but all of these methods should make sense for it. We don't need to gather all the methods in one place: it's actually better to make several interfaces instead. Whatever decisions you make while writing, remember to try hard and create a better world in your code!

What do classes need to do to implement an interface?

Implement all methods from it